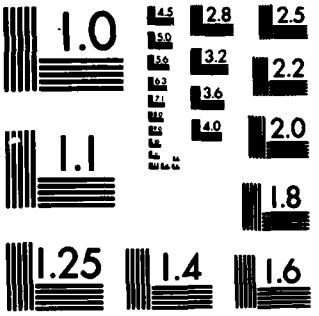
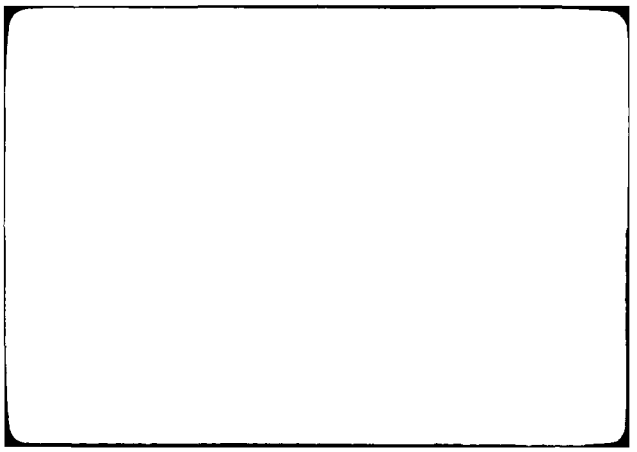
AD-A125 290 ANOMALIES IN PARALLEL BRANCH-AND-BOUND ALGORITHMS(U) 1/1
MINNESOTA UNIV MINNEAPOLIS DEPT OF COMPUTER SCIENCE
T LAI ET AL. DEC 82 TR-82-25 N00014-80-C-0650

UNCLASSIFIED F/G 12/1 NL

END
DATE FILMED
83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



Computer Science Department
136 Lind Hall
Institute of Technology
University of Minnesota
Minneapolis, Minnesota 55455

(2)

Anomalies In Parallel Branch-and-Bound Algorithms

by

Ten-Hwang Lai and Sartaj Sahni

Technical Report 82-25

December 1982

Anomalies in Parallel Branch-and-Bound Algorithms*

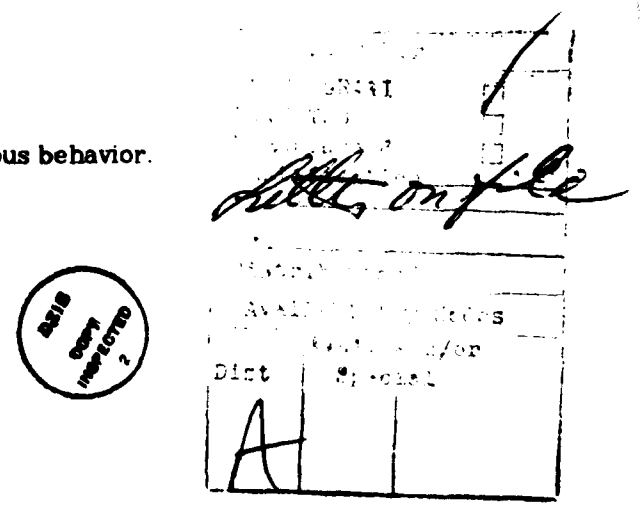
Ten-Hwang Lai and Sartaj Sahni
University of Minnesota

Abstract

We consider the effects of parallelizing branch-and-bound algorithms by expanding several live nodes simultaneously. It is shown that it is quite possible for a parallel branch-and-bound algorithm using n_2 processors to take more time than one using n_1 processors even though $n_1 < n_2$. Furthermore, it is also possible to achieve speedups that are in excess of the ratio n_2/n_1. Experimental results with the 0/1 Knapsack and Traveling Salesperson problems are also presented.

Key Words and Phrases

Parallel Algorithms, branch-and-bound, anomalous behavior.



* This research was supported in part by the Office of Naval Research under contract N00014-80-C-0650.

1. Introduction

Branch-and-bound is a popular algorithm design technique that has been successfully used in the solution of problems that arise in various fields (e.g., combinatorial optimization, artificial intelligence, etc.) [1, 6 - 12]. We shall briefly describe the branch-and-bound method as used in the solution of combinatorial optimization problems. Our terminology is from Horowiz and Sahni [7].

In a combinatorial optimization problem we are required to find a vector x $= (x_1, x_2, \ldots, x_n)$ that optimizes some criterion function f(x) subject to a set C of constraints. This constraint set may be partioned into two subsets: explicit and implicit constraints. Implicit constraints specify how the x_is must relate to each other. Two examples are:

1) $\sum_{i=1}^{n} a_i x_i \leq b$

2) $a_1 x_1^2 - a_2 x_1 x_2 + a_3 x_3^4 = 6$

Explicit constraints specify the range of values each x_i can take. For example:

1) $x_i \in \{0, 1\}$

2) $x_i \geq 0$

The set of vectors that satisfy the explicit constraints defines the ***solution space***. In a branch-and-bound approach this solution space is organized as a graph which is usually a tree. This resulting organization is called a ***state space graph (tree)***. All the state space graphs used in this paper are ***trees***. So we shall henceforth only refer to state space trees. Figure 1 shows a state space tree for the case n = 3 and $x_i \in \{0, 1\}$. The path from the root to some of the nodes (in this case the leaves) defines an element of the solution space. Nodes with this property are called ***solution nodes***. Solution nodes that satisfy the implicit constraints are called ***feasible solution nodes*** or ***answer nodes***. Answer nodes have been drawn as double circles in Figure 1. The *cost* of an answer node is the value of the criterion function at that node. In solving a combinatorial optimization problem we wish to find a ***least cost answer node***.

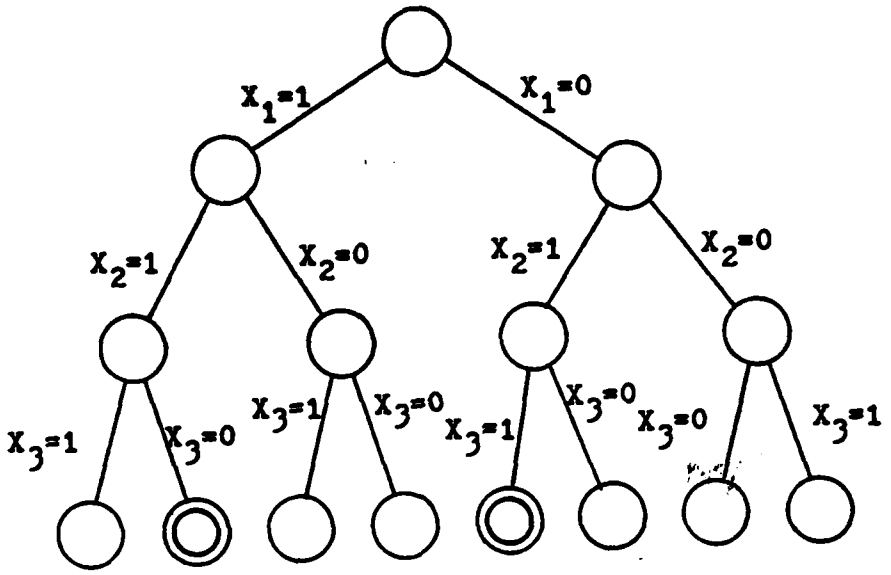


Figure 1 A state space tree

For convenience we assume that we wish to minimize f(x). With every node N in the state space tree, we associate a value $f_{min}(N) = \min\{f(Q) : Q \text{ is a feasible solution node in the subtree N}\}$. (If there exists no such Q, then let $f_{min}(N) = \infty$.)

While there are several types of branch-and-bound algorithms, we shall be concerned only with the more popular ***least cost branch-and-bound*** (lcbb). In this method a heuristic function g( ) with the following properties is used:

(P1) $g(N) \leq f_{min}(N)$ for every node N in the state space tree.

(P2) g(N) = f(N) for solution nodes representing feasible solutions (i.e., answer nodes).

(P3) $g(N) = \infty$ for solution nodes representing infeasible solutions.

(P4) $g(N) \geq g(P)$ if N is a child of P.

g( ) is called a ***bounding function***. lcbb generates the nodes in a state space tree using g( ). A node that has been genereated, can lead to a feasible solution, and whose children haven't yet been generated is called a ***live node***. A list of live nodes (generally as a heap) is maintained. In each iteration of the lcbb a live node, N, with least g( ) value is selected. This node is called the current ***E-node***. If N is an answer node, it must be a least cost answer node. If N is not an answer node, its children are generated. Children that cannot lead to a least cost answer node (as determined by some heuristic) are discarded. The remaining children are added to the list of live nodes.

The problem of parallelizing lcbb has been studied earlier [2 - 5, 13]. There are essentially three ways to introduce parallelism into lcbb:

(1) Expand more than 1 E-node during each iteration.

(2) Evaluate g( ) and determine feasibility in parallel.

(3) Use parallelism in the selection of the next E-node(s).

Wah and Ma [13] exclusively consider (1) above (though they point out (2) and (3) as possible sources of parallelism). If p processors are available then q = min{p, number of live nodes} live nodes are selected as the next set of E-nodes (these are the q live nodes with smallest g( ) values). Let g_{min} be the least g value among these q nodes. If any of these E-nodes is an answer node and has g( ) value equal to g_{min} then a least cost answer node has been found. Otherwise all q E-nodes are expanded and their children added to the list of live nodes. Each such expansion of q E-node counts as one iteration of the parallel lcbb. For any given problem instance and g, let I(p) denote the number of iterations needed when p processors are available. Intuition suggests that the following might be true about I(p):

(I1) $I(n_1) \geq I(n_2)$ whenever $n_1 < n_2$

(I2) $\frac{I(n_1)}{I(n_2)} \leq \frac{n_2}{n_1}$

In Section 2, we show that neither of these two relations is in fact valid. Even if the g( )s are restricted beyond (P1) - (P4), these relations do not hold. The experimental results provided in Section 3 do, however, show that (I1) and (I2) can be expected to hold "most" of the time.

Wah and Ma [13] experimented with the vertex cover problem using 2^k, $0 \leq k \leq 6$ processor. Their results indicate that $I(1)/I(p) \simeq p$. Our experiments with the 0/1-Knapsack and Traveling Salesperson problems indicate that $I(1)/I(p) \simeq p$ only for "small" values of p (say $p \leq 16$).

2. Some Theorems For Parallel Branch-and-Bound

As remarked in the introduction, several anomalies occur when one parallelizes branch-and-bound algorithms by using several E-nodes at each iteration. In this section we establish these anomalies under varying constraints for the bounding function g( ). First, it should be recalled that the g( ) functions typically used (eg. for the knapsack problem, traveling salesperson problem, etc. cf. [7]) have the following properties:

(a) $g(N) \geq g(M)$ whenever N is a child of node M. Thus, the g( ) values along any path from the root to a leaf form a nondecreasing sequence.

(b) Several nodes in the state space tree may have the same g( ) value. In fact, many nonsolution nodes may have a g( ) value equal to f^*. This is particularly true of nodes that are near ancestors of solution nodes.

In constructing example state space trees, we shall keep (a) in mind. None of the trees constructed will violate (a) and we shall not explictly make this point in further discussion. The first result we shall establish is that it is quite possible for a parallel branch-and-bound using n_2 processors to perform much worse than one using a fewer number n_1 of processors.

Theorem 1: Let $n_1 < n_2$. For any $k > 0$, there exists a problem instance such that $kI(n_1) < I(n_2)$.

Proof: Consider a problem instance with the state space tree of Figure 2. All nonleaf nodes have the same g( ) value equal to f^*, the f value of the least cost answer node (node A). When n_1 processors are available, one processor expands the root and generates its $n_1 + 1$ children. Let us suppose that on iteration 2, the left n_1 nodes on level 2 get expanded. Of the n_1 children generated $n_1 - 1$

get bounded and only one remains live. On iteration 3 the remaining live node on level 2 (B) and the one on level 3 are expanded. The level 3 node leads to the solution node and the algorithm terminates with $I(n_1) = 3$.

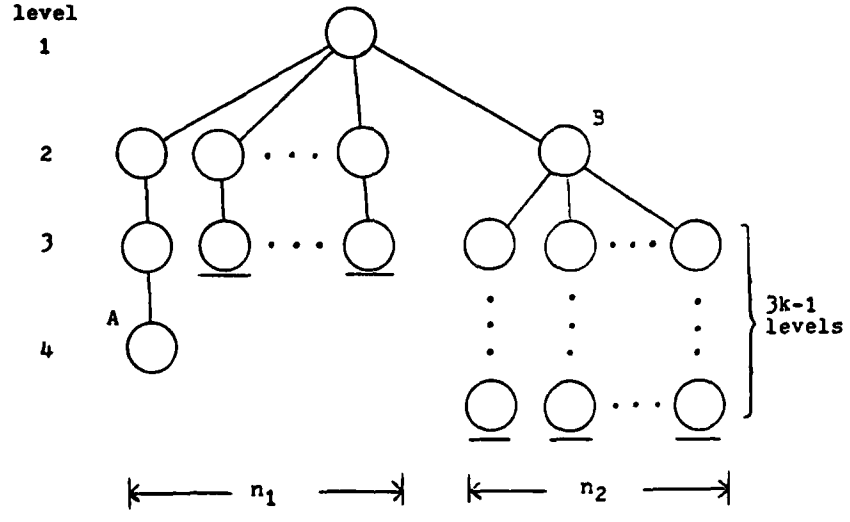


Figure 2: Instance for Theorem 1

When n_2 processors are available, the root is expanded on iteration 1 and all $n_1 + 1$ live nodes from level 2 get expanded on iteration 2. The result is $n_2 + 1$ live nodes on level 3. Of these, only n_2 can be expanded on iteration 3. These n_2 could well be the rightmost n_2 nodes. And iterations 4, 5, ..., 3k could very well be limited to the rightmost subtree of the root. Finally in iteration 3k + 1, the least cost answer node a is generated. Hence, $I(n_2) = 3k + 1$ and $kI(n_1) < I(n_2)$. []

In the above construction, all nodes have the same $g(\ )$ value, f^*. While this might seem extreme, property (b) above states that it is not unusual for real g-

functions to have a value f^* at many nodes. The example of Figure 2 does serve to illustrate why the use of additional processors may not always be rewarding. The use of an additional processor can lead to the development of a node N (such as node B of Figure 2) that looks "promising" and eventually diverts all or a significant number of the processors into its subtree. When a fewer number of processors are used, the upper bound U at the time this "promising" node is to get expanded might be such that $U \leq g(N)$ and so N is not expanded when a fewer number of processors are available.

The proof of Theorem 1 hinges on the fact that g(N) may equal f^* for many nodes (independent of whether these nodes are least cost answer nodes or not). If we require the use of g-functions that can have the value f^* only for least cost answer nodes, then Theorem 1 is no longer valid for all combinations of n_1 and n_2, $n_1 < n_2$. In particular, if $n_1 = 1$ then the use of more processors never increases the number of iterations (Theorem 2).

Definition: A node N is *critical* iff $g(N) < f^*$.

Theorem 2: If $g(N) \neq f^*$ whenever N is not a least cost answer node, then $I(1) \geq I(n)$ for $n > 1$.

Proof: When the number of processors is 1, only critical nodes and least cost answer nodes can become E-nodes (as whenever an E-node is to be selected there is at least one node N with $g(N) \leq f^*$ in the list of live nodes). Furthermore, every critical node becomes an E-node by the time the branch-and-bound algorithm terminates. Hence, if the number of critical nodes is m, $I(1) = m$.

When $n > 1$ processors are available, some noncritical nodes may become E-nodes. However, at each iteration, at least one of the E-nodes must be a critical node. So, $I(n) \leq m$. Hence, $I(1) \geq I(n)$. []

When $n_1 \neq 1$, a degradation in performance is possible with $n_2 > n_1$ even if we restrict the g( )s as in Theorem 2.

Theorem 3: Assume that $g(N) \neq f^*$ whenever N is not a least cost answer node. Let $1 < n_1 < n_2$ and $k > 0$. There exists a problem instance such that $I(n_1) + k \leq$

$I(n_2)$.

Proof: Figures 3(a) and 3(b) show two identical subtrees T. Assume that all nodes have the same g( ) value and are critical. The numbers inside each node give the iteration number in which that node becomes an E-node when n_1 processors are used (Figure 3(a)) and when n_2 processors are used (Figure 3(b)). Other evaluation orders are possible. However, the ones shown in Figures 3(a) and 3(b) will lead to a proof of this theorem.

We can construct a larger state space tree by connecting together k copies of T (Figure 3(c)). The B node of one copy connects to the A node (root) of the next. Each triangle in this figure represents a copy of T. The least cost answer node is the child of the B node of the last copy of T. It is clear that for the state space tree of Figure 3(c), $I(n_1) = jk$ while $I(n_2) = (j + 1)k$. Hence, $I(n_1) + k = I(n_2)$. []

The assumption that $g(N) \neq f^*$ when N is not a least cost answer node is not too unrealistic as it is often possible to modify typical g( )s so that they satisfy this requirement. The example of Figure 3 has many nodes with the same g( ) value and so we might wonder what would happen if we restricted the g( )s so that only least cost answer nodes can have the same g( ) value. This restriction on g( ) is quite severe and, in practice, it is often not possible to guarantee that the g( ) in use satisfies this restriction. However, despite the severity of the restriction one cannot guarantee that there will be no degradation of performance using n_2 processors when $n_1 < n_2 < 2(n_1 - 1)$. We have unfortunately been unable to extend our result of Theorem 4 to the case when $n_2 \geq 2(n_1 - 1)$. So, it is quite possible that no degradation is possible when the number of processors is (approximately) doubled and g( ) is restricted as above.

Theorem 4: Let $n_1 < n_2 < 2(n_1 - 1)$ and let $k > 0$. There exists a g( ) and a problem instance that satisfy the following properties:

(a) $g(N_1) \neq g(N_2)$ unless both of N_1 and N_2 are least cost answer nodes.

(b) $I(n_1) + k \leq I(n_2)$.

Proof: Consider the state space tree of Figure 4(a). The number outside each node is its g( ) value while the number inside a node gives the iteration in which

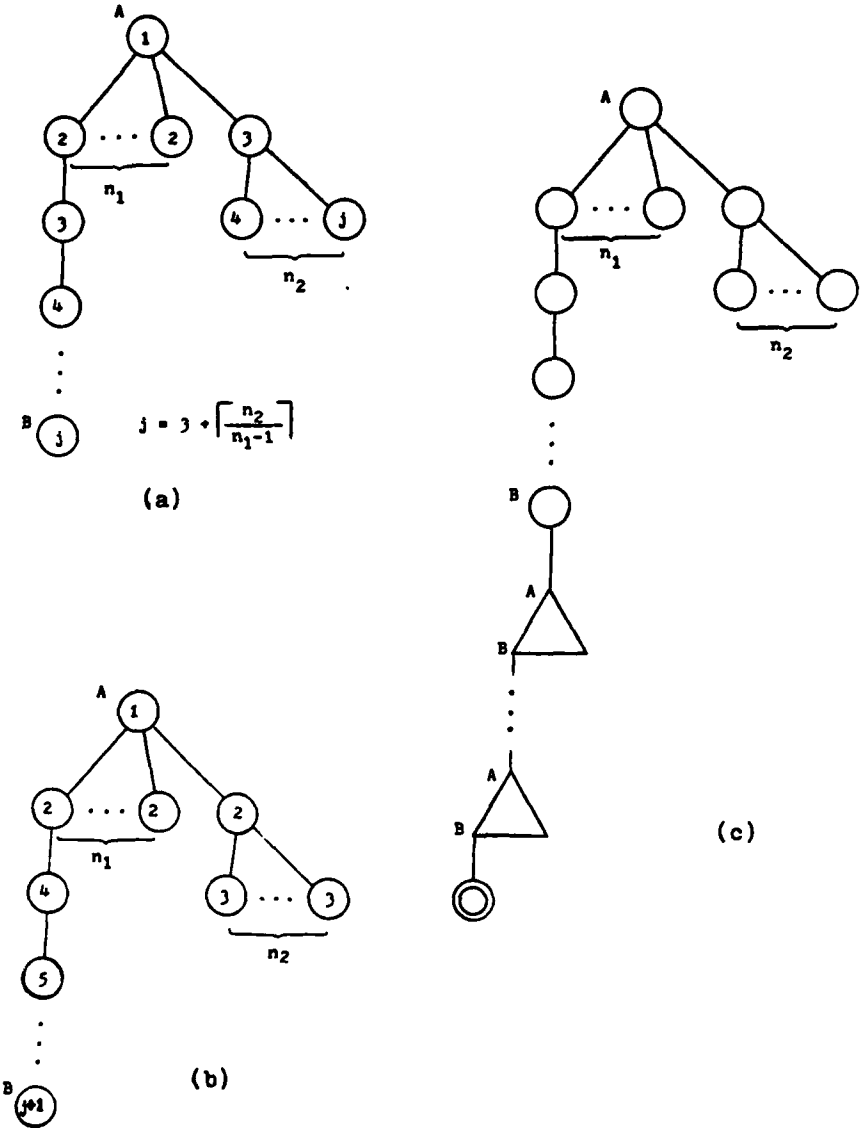


Figure 3: Instance for Theorem 3

that node is the E-node when n_1 processors are used. It takes n_1 processors 4 iterations to get to and evaluate node B. When n_2 processors are available, $n_1 < n_2 < 2(n_1 - 1)$, the iteration numbers are as given in Figure 4(b). This time 5 iterations are needed. Combining k copies of this tree and setting the g( )

values in each copy to be different from those in other copies yields the tree of Figure 4(c). For this tree, we see that $I(n_1) + k = I(n_2)$. []

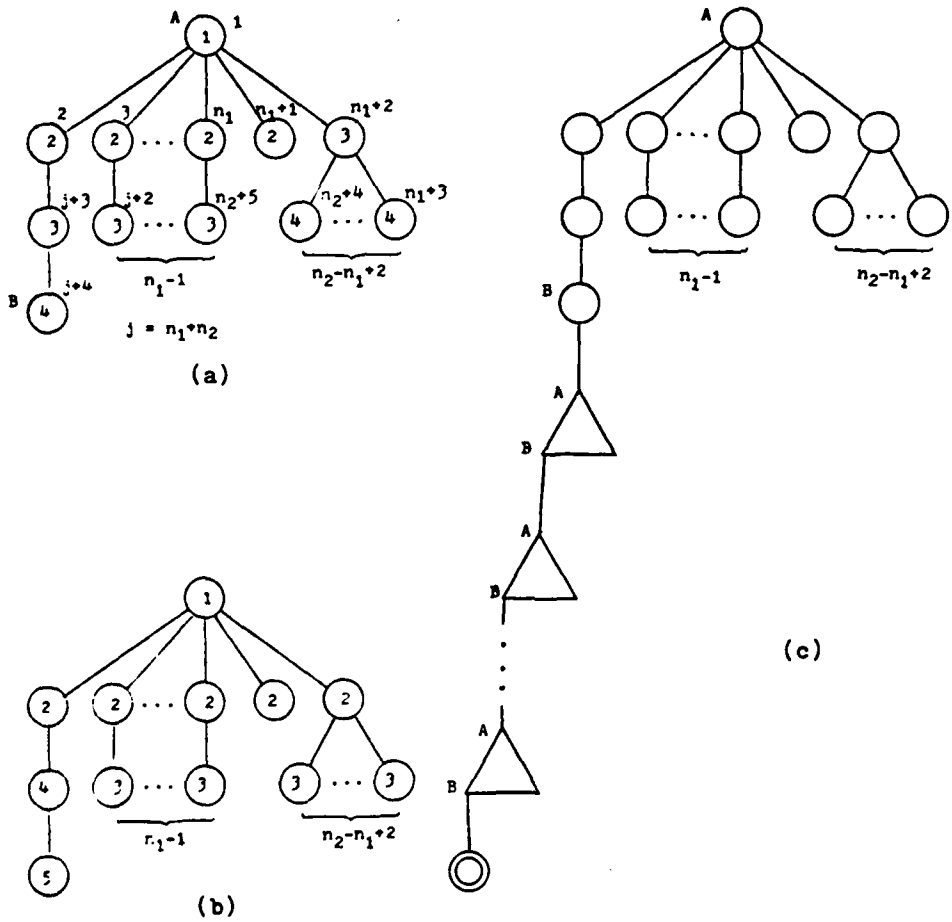


Figure 4: Instance for Theorem 4

The remaining results we shall establish in this section are concerned with the maximum improvement in performance one can get in going from n_1 to n_2 processors, $n_1 < n_2$. Generally, one would expect that the performance can increase by at most n_2 / n_1. This is not true for branch-and-bound. In fact, Theorem 5 shows that using $g(\ )$s that satisfy properties (a) and (b), an unbounded improvement in performance is possible. The reason for this is much the same as for the possibility of an unbounded loss in performance. The additional processors might enable us to improve the upper bound quickly thereby curtailing the expansion of some of the nodes that might get expanded without these processors.

Theorem 5: Let $n_1 < n_2$. For any $k > n_2/n_1$, there exists a problem instance for which $I(n_1)/I(n_2) \geq k > n_2/n_1$.

Proof: Simply consider the state space tree of Figure 5(a). All nodes have the same $g(\ )$ value, f^*. Assume that when n_1 processors are used, the n_1 nodes at the left end of level i become E-nodes on iteration i, $2 \leq i \leq 2k$. Hence, $I(n_1) = 2k + 1$. When $n_2 > n_1$ processors are used, $I(n_2) = 2$ (Figure 5(b)) and $I(n_1)/I(n_2) > k$. []

As in the case of Theorem 2, we can show that when $g(N) \neq f^*$ whenever N is not a least cost answer node, $I(1)/I(n) \leq n$.

Theorem 6: Assume that $g(N) \neq f^*$ whenever N is not a least cost answer node. $I(1)/I(n) \leq n$ for $n > 1$.

Proof: From the proof of Theorem 2, we know that $I(1) = m$ where m is the number of critical nodes. Since all critical nodes must become E-nodes before the branch-and-bound algorithm can terminate, $I(n) \geq m/n$. Hence, $I(1)/I(n) \leq n$. []

When $1 < n_1 < n_2$ and g(N) is restricted as above, $I(n_1)/I(n_2)$ can exceed n_2/n_1 but cannot exceed n_2.

Theorem 7: Assume that $g(N) \neq f^*$ whenever N is not a least cost answer node. Let $1 < n_1 < n_2$. The following are true:

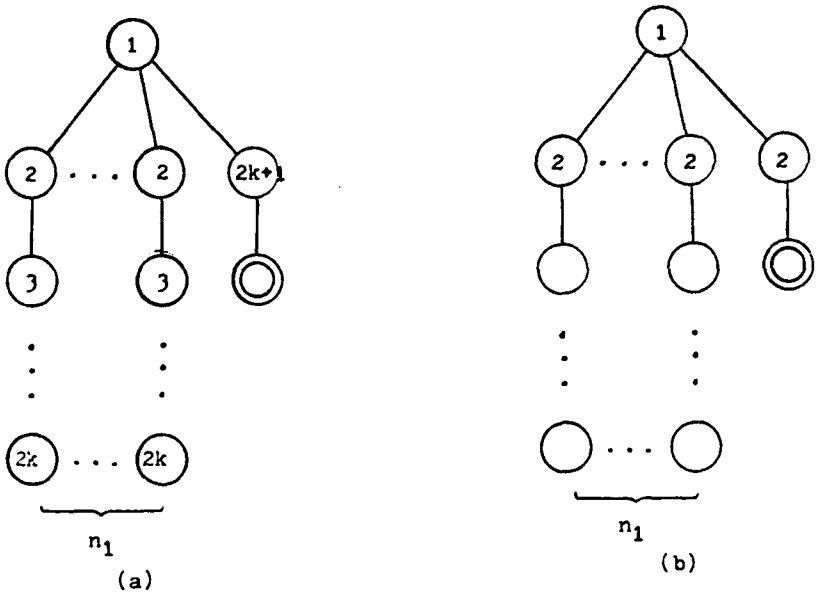


Figure 5: Instance for Theorem 5

(1) $I(n_1)/I(n_2) \leq n_2$.

(2) There exists a problem instance for which $I(n_1)/I(n_2) > n_2/n_1$.

Proof: (1) From Theorems 2 and 6, we immediately obtain:

$$\frac{I(n_1)}{I(n_2)} = \frac{I(n_1)}{I(1)} \cdot \frac{I(1)}{I(n_2)} \leq n_2.$$

(2) For simplicity, we assume that $k = n_2 / n_1$ is an integer. Consider the state space tree of Figure 6. The g( ) value of all nodes other than the one representing the least cost answer is less than f^*. The number inside (outside) a node is the iteration in which it is the E-node when n_1 (n_2) processors are used. We see that $I(n_1) = n_2(k + 1) + 1$ and $I(n_2) = n_2 + 2$. Hence,

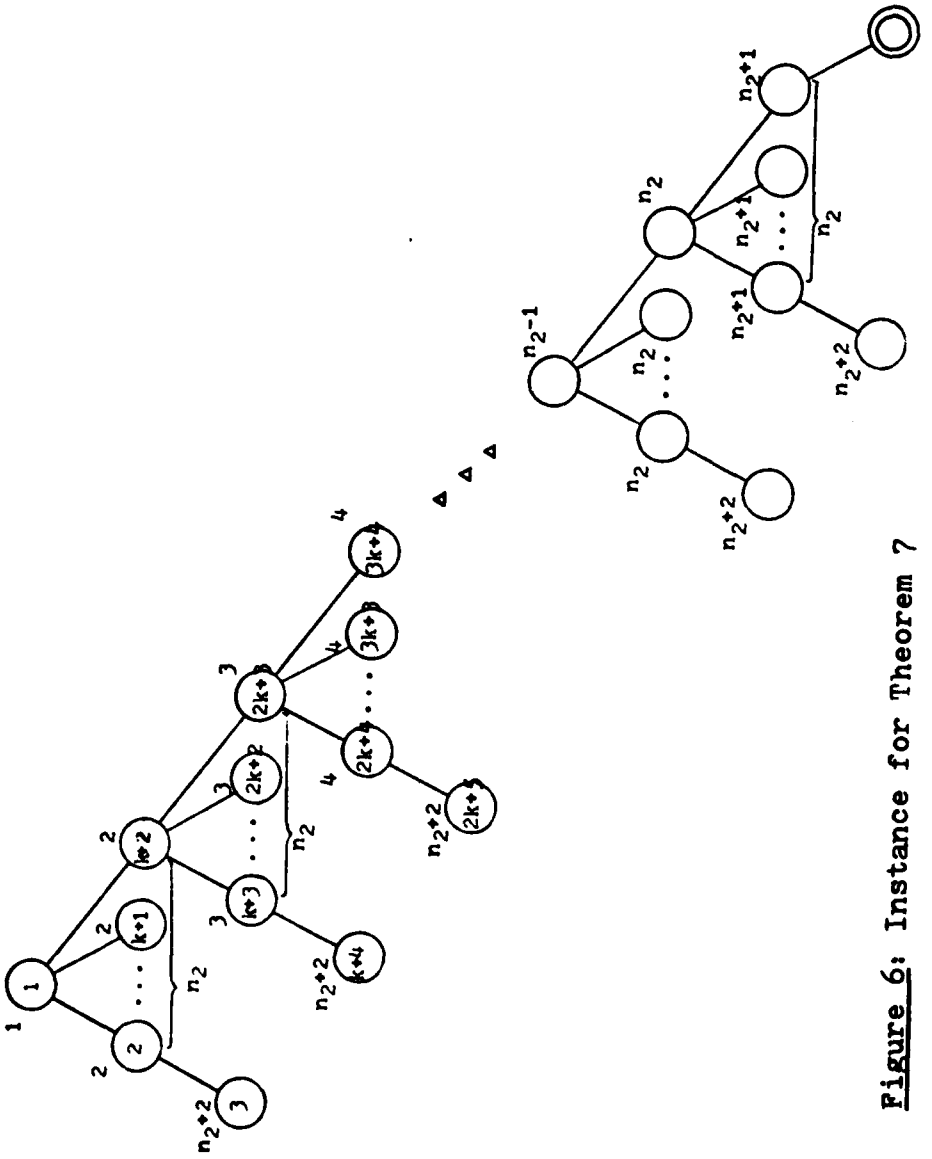


Figure 6: Instance for Theorem 7

$$\frac{I(n_1)}{I(n_2)} = \frac{n_2 k + n_2 + 1}{n_2 + 2} > k. \quad []$$

3. Experimental Results

In order to determine the frequency of anomalous behavior described in the previous section, we simulated a parallel branch-and-bound with 2^k processors for k = 0, 1, 2, ..., 9. Two test problems were used: 0/1-Knapsack and Traveling Salesperson. These are described below.

0/1-Knapsack:

In this problem we are given n objects and a knapsack with capacity M. Object i has associated with it a profit p_i and a weight w_i. We wish to place a subset of the n objects into the knapsack such that the knapsack capacity is not exceeded and the sum of the profits of the objects in the knapsack is maximum. Formally, we wish to solve the following problem:

$$\text{maximize } \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \le M,\ x_i \in \{0, 1\}.$$

Horowitz and Sahni [7] describe two state space trees that could be used to solve this problem. One results from what they call the fixed tuple size formulation. This is a binary tree such as the one shown in Figure 7(a) for the case n = 3. The other results from the variable tuple size formulation This is an n-ary tree. When n = 3, the resulting tree is as in Figure 7(b). The bounding function used is the same as the one described in [7]. Since the bounding function requires that objects be ordered such that $p_i / w_i \ge p_{i+1} / w_{i+1}$, $1 \le i < n$, we generated our test data by first generating random w_is. The p_is were then computed from the w_is by using a random nonincreasing sequence $f_1, f_2, \ldots, f_n$ and the equation $p_i = f_i w_i$. We generated 100 instances with n = 50 and 60 instances with n = 100. These 160 instances were solved using the binary state space tree described above. (We also tried the n-ary state space tree but found that it would take several weeks of computer time to complete our simulation. The reason it will take so much time is that when n-ary state space trees are used a great number of nodes will be generated and the queue of live nodes will exceed the capacity of main memory and has to be moved to the secondary storage. In our program, it is time consumming to maintain a queue of live

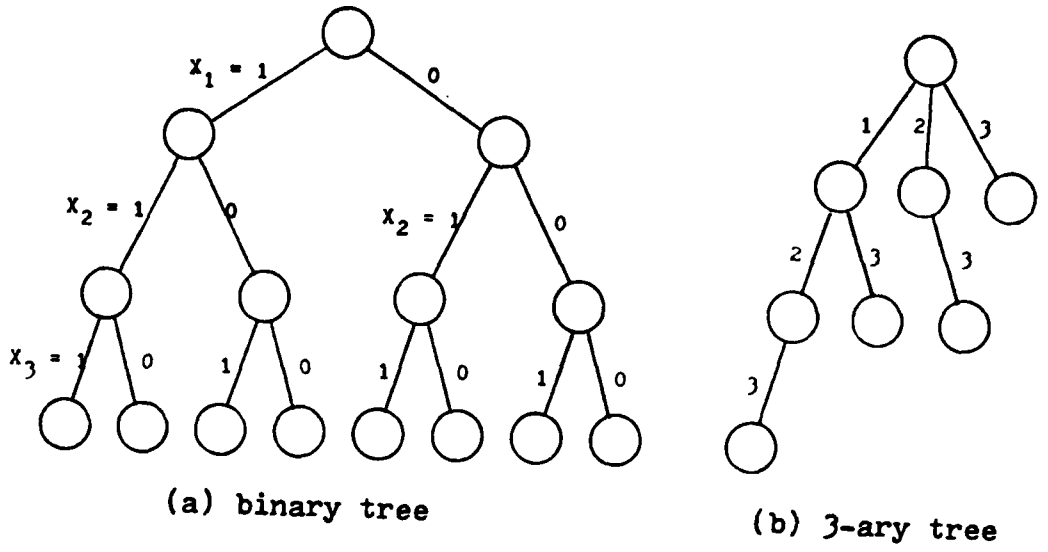


Figure 7

nodes that must be partly stored in secondary storage.)

Table 1 gives the average values for I(p), I(1)/I(p) and I(p)/I(2p). From Table 1, we see that when n = 50, I(1)/I(p) is significantly less than p for p > 2. The observed improvement in performance is not as high as one might expect. Similarly, the ratio I(p)/I(2p) drops rapidly to 1 and is acceptable only for p = 1 and 2 (see also Figure 8). In none of the 100 instances tried for n = 50 did we observe anomalous behavior. I.e., it was never the case that I(p) < I(2p) or that I(p) > 2I(2p).

When n = 100, the ratio I(1)/I(p) is significantly less than p for p > 8 (see also Figure 9). Of the 60 instances run, 6 (or 10%) exhibited anomalous behavior. For all 6 of these there was at least one p for which I(p) > 2I(2p). There was only one case where I(p) < I(2p). The values of I(p), I(1)/I(p), and I(p)/I(2p) for these

	n = 50			n = 100		
p	I(p)	$\frac{I(1)}{I(p)}$	$\frac{I(p)}{I(2p)}$	I(p)	$\frac{I(1)}{I(p)}$	$\frac{I(p)}{I(2p)}$
1	363	1.00	1.87	2814	1.00	2.19
2	188	1.87	1.68	1351	2.19	1.85
4	106	3.17	1.42	754	3.69	1.75
8	70	4.66	1.22	402	6.47	1.60
16	56	5.97	1.09	232	10.58	1.35
32	51	6.84	1.03	162	14.94	1.22
64	50	7.23	1.00	126	19.35	1.14
128	50	7.26	1.00	108	23.68	1.05
256	50	7.26	1.00	102	25.84	1.02
512	50	7.26	1.00	100	27.06	1.01
1024	50	7.26		100	27.68	

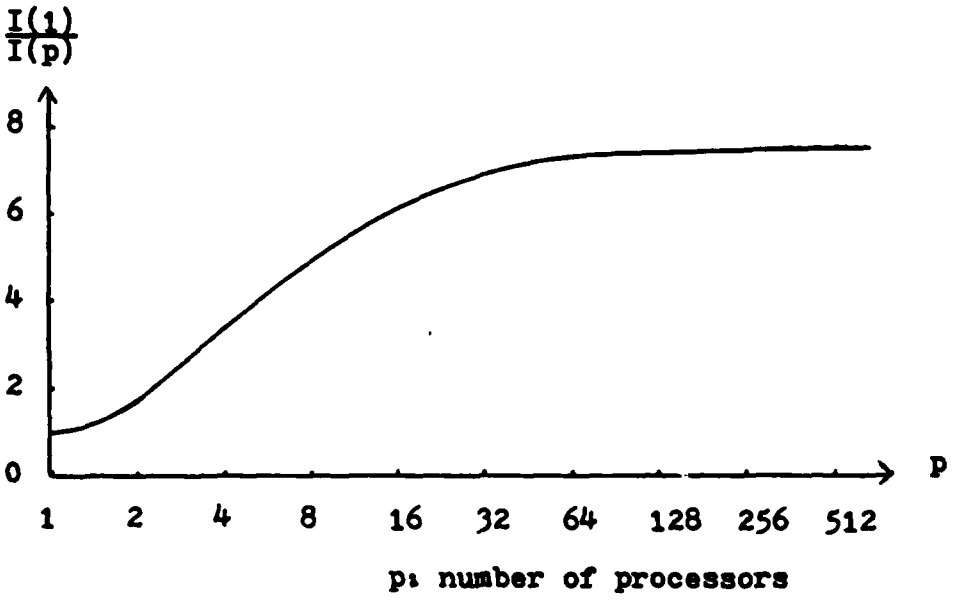
Table 1: Experimental results (knapsack)

six instances is given in Table 2. It is striking to note the instance for which I(1)/I(2) = 14.6 and I(2)/I(4) = 0.15.

The Traveling Salesperson Problem:

Here we are given an n vertex undirected complete graph. Each edge is assigned a weight. A *tour* is a cycle that includes every vertex (i.e., it is a Hamiltonian cycle). The *cost* of a tour is the sum of the weights of the edges on the tour. We wish to find a tour of minimum cost.

The branch-and-bound strategy that we used is a simplified version of the one proposed by Held and Karp [6]. Vertex 1 is chosen as the start vertex. There are n - 1 possibilities for the next vertex and n - 2 for the preceding vertex (assume n > 2). This leads to (n - 1)(n - 2) sequences of 3 vertices each. Half of these may be discarded as they are symmetric to other sequences. Any sequence with an edge having infinite weight may also be discarded. Paths are expanded one vertex at a time using the set of vertices adjacent to the end of the path. A lower bound for the path $(i_1, i_2, \ldots, i_k)$ is obtained by computing the cost of the minimum spanning tree for $\{1, 2, \ldots, n\} - \{i_1, i_2, \ldots, i_k\}$ and adding an edge from each of i_1, and i_k to this spanning tree in such a way that these edges connect to the two nearest vertices in the spanning tree.



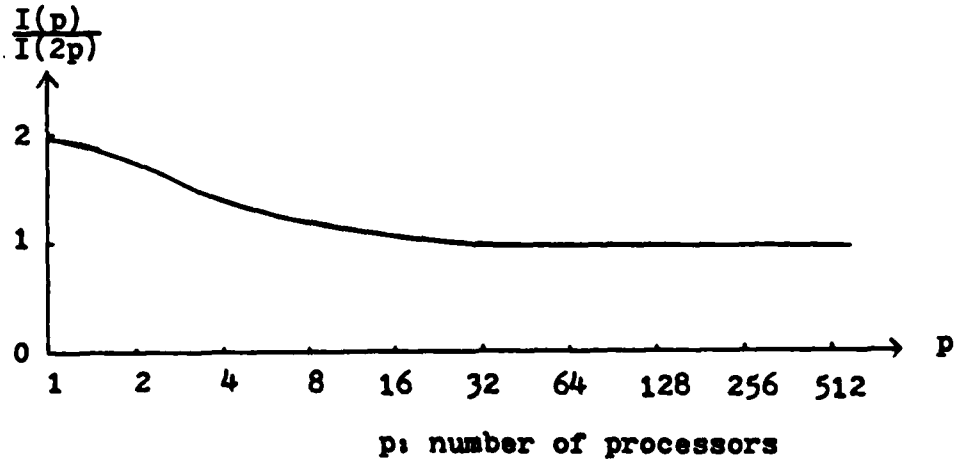


Figure 8: Knapsack with 50 objects

In our experiment with the traveling salesperson problem we generated 45 instances each having 20 vertices. The weights were assigned randomly. However, each edge had a finite weight with probability 0.35. Use of a much higher probability results in instances that take years of computer time to solve by the

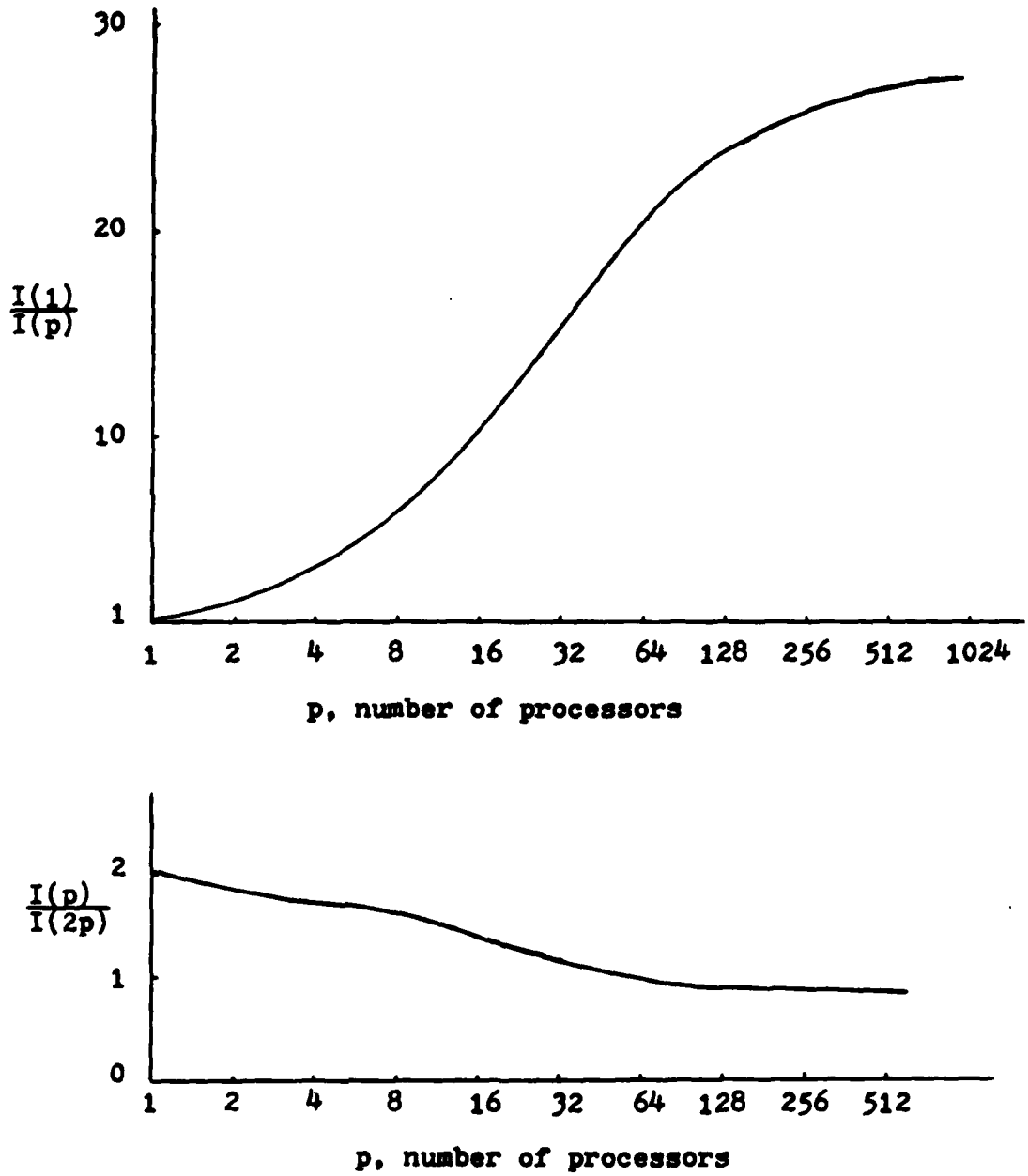


Figure 9: Knapsack with 100 objects

branch-and-bound method.

p	I(p)	$\frac{I(1)}{I(p)}$	$\frac{I(p)}{I(2p)}$
1	2131	1.00	1.79
2	1191	1.79	**2.23**
4	533	4.00	1.49
8	357	5.97	**2.01**
16	178	11.97	1.09
1	1009	1.00	**2.19**
2	461	**2.19**	1.57
1	21593	1.00	1.99
8	3060	7.06	**2.04**
16	1503	14.37	1.81
1	4119	1.00	**2.03**
2	2034	**2.03**	**2.06**
4	987	**4.17**	1.84
1	5251	1.00	**3.04**
2	1725	**3.04**	1.90
4	909	**5.78**	1.41
8	646	**8.13**	1.74
16	372	14.12	**2.01**
32	185	28.38	1.42
1	7510	1.00	**14.64**
2	513	**14.64**	**0.15**
4	3346	**2.24**	**2.86**
8	1174	6.40	**2.23**
16	527	14.25	**0.95**
32	552	13.61	1.71

Table 2: Data exhibiting anomalous behavior

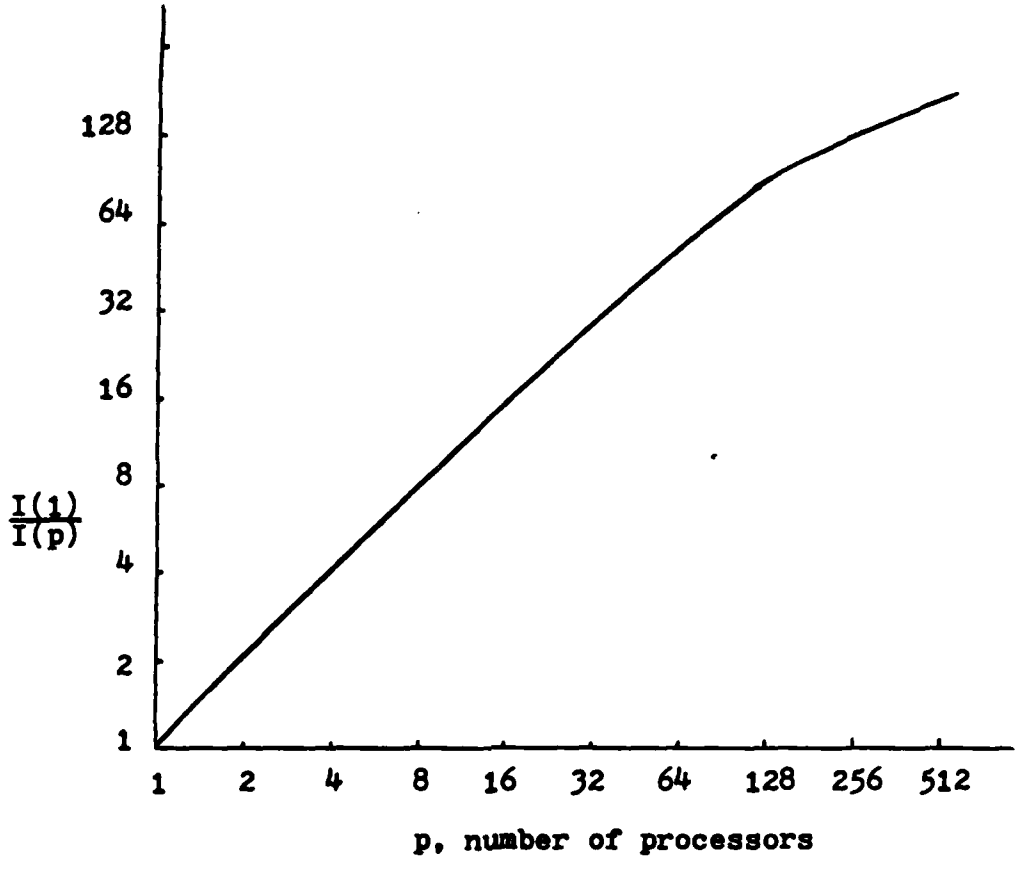
Those 45 instances were solved using $p = 2^k$, $0 \leq k \leq 9$ processors. The average values of I(p), I(1)/I(p), and I(p)/I(2p) are tabulated in Table 3. As can be seen, for $p \leq 32$ the average value of I(1)/I(p) is quite close to p and the average value of I(p)/I(2p) is quite close to 2 (see also Figure 10). No anomalies were observed for any of these 45 instances.

p	I(p)	$\frac{I(1)}{I(p)}$	$\frac{I(p)}{I(2p)}$
1	3974	1.000	1.996
2	1989	1.996	1.990
4	996	3.973	1.976
8	500	7.849	1.943
16	252	15.258	1.873
32	129	28.685	1.753
64	68	51.126	1.609
128	39	85.378	1.417
256	25	129.411	1.252
512	19	177.459	

Table 3: Experimental results (traveling salesperson)

4. Conclusions

We have demonstrated the existence of anomalous behavior in parallel branch-and-bound. *Our experimental results indicate that such anomalous behavior will* be rarely witnessed in practice. Furthermore, there is little advantage to expanding more than k nodes in parallel. k will in general depend on both the problem and the problem size being solved. If we require I(p)/I(2p) to be at least 1.66, then for the knacksack problem with n = 50, k is between 4 and 8 whereas with n =100 it is between 8 and 16 (based on our experimental results). For the traveling salesperson problem with 20 vertices k is between 8 and 16. *If* p is larger than k, then more effective use of the processors is made when they are divided into k groups each of size approximately p/k. Each group of processors is used to expand a single E-node in parallel. If s is the speedup obtained by expanding an E-node using q processors, then allocating q processors to each E-node and expanding only p/q E-nodes in parallel is preferable to expanding p E-nodes in parallel provided that sI(1)/I(p/q) > I(1)/I(p).



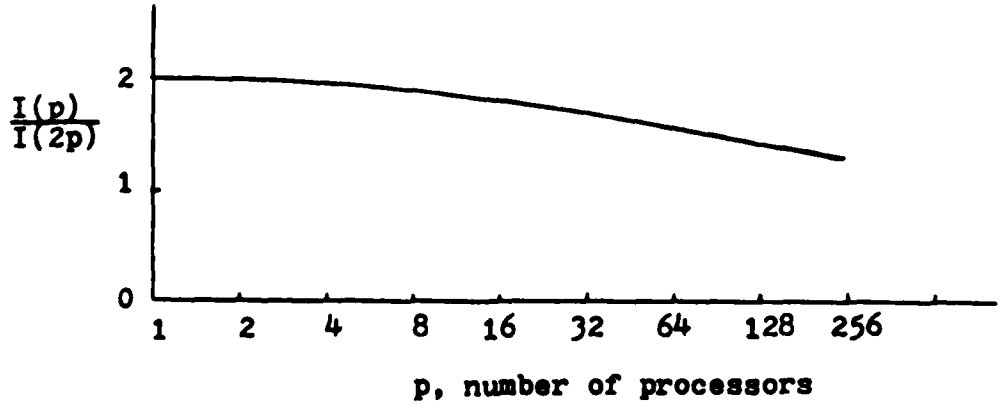


Figure 10: Traveling Salesperson

References

1. N. Agin, "Optimum seeking with branch-and bound," *Manage. Sci.*, Vol. 13, pp. B176-B185.
2. B. Desai, "The BPU, a staged parallel processing system to solve the zero-one problem," *Proceedings of ICS '78*, 1978, pp. 802-817.
3. B. Desai, "A parallel microprocessing system," *Proceedings of the 1979 International Conference on Parallel Processing*, 1979.
4. O. El-Dessouki and W. Huen, "Distributed enumeration on network computers," *IEEE Transactions on Computers*, C-29, 1980, pp. 818-825.
5. J. Harris and D. Smith, "Hierarchical multiprocessor organizations," *Proceedings of the 4th Annual Symposium on Computer Architecture*, 1977, pp. 41-48.
6. M. Held and R Karp, "The traveling salesman problem and minimum spanning trees: part II," *Math Prog.*, 1, pp. 6-25, 1971.
7. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Inc., 1978.
8. E. Ignall and L.Schrage, "Application of the branch-and-bound technique to some flow-shop scheduling problems," *Oper. Res., 13, pp. 400-412, 1965.*
9. W. Kohler and K. Steiglitz, "Enumerative and iterative computational approaches," in E. Coffman (ed.) *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, Inc., New York, 1976, pp. 229-287.
10. E. Lawer and D. Wood, "Branch-and bound methods: a survey," *Oper. Res.*, 14, pp. 699-719, 1966.
11. L. Mitten, "Branch-and-bound methods: general formulation and properties," *Oper. Res.*, 18, pp. 24-34, 1970.
12. N. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.
13. B. Wah and Y. Ma, "NANIP - a parallel computer system for implementing branch-and-bound algorithm," *Proceedings of The 8th Annual Symposium on Computer Architecture*, 1982, pp. 239-262.


DA
FIL